Hierarchical State Machines as Modular Horn Clauses *

Pierre-Loïc Garoche
DTIM, UFT, Onera – The French Aerospace Lab

Temesghen Kahsai
NASA Ames / CMU

Xavier Thirioux
IRIT/ENSEEIHT, UFT, CNRS

In model based development, embedded systems are modeled using a mix of dataflow formalism, that capture the flow of computation, and hierarchical state machines, that capture the modal beahviour of the system. For safety analysis, existing approaches rely on a compilation scheme that transform the original model (dataflow and state machines) into a pure dataflow formalism. Such compilation often result in loss of important structural information that capture the modal behaviour of the system. In previous work we have developed a compilation technique from a dataflow formalism into modular Horn clauses. In this paper, we present a novel technique that faithfully compile hierarchical state machines into modular Horn clauses. Our compilation technique preserves the structural and modal behavior of the system, making the safety analysis of such models more tractable.

1 Introduction

Model-based development is a leading technique in developing software for critical embedded systems such as automotive, avionics systems, train controllers and medical devices. Typically such systems are modeled using a mix of *dataflow* formalism and *hierarchical state machines*. For instance, Matlab Simulink [20] or Esterel SCADE [9] diagrams are typically used to specify aspects of a system that can be modeled by differential equations relating inputs and outputs (i.e., dataflow), while Matlab Stateflow [21] charts or Esterel SCADE automata usually model the control aspects. The extensive use of the aforementioned formalism in the development of safety-critical systems, associated with certification standards [8] that recommend the use of formal methods for the specification, design, development and verification of software, makes a formal treatment of these notations extremely crucial.

For the purpose of safety analysis, Simulink/SCADE models are compiled to a lower level modeling language, usually a synchronous dataflow language such as Lustre [5]. Preserving the original (hierarchical and modular) structure of the model is paramount to the success of the analysis process. In [10] we have illustrated a technique to preserve such structure via a modular compilation process. Specifically, we presented a technique that consists of compiling in a modular fashion Lustre programs into constrained Horn clauses. In this paper, we extend our previous compilation schema to handle hierarchical state machines (i.e. Stateflow diagrams or SCADE automata). Hierarchical state machines allows to capture the complex *modal* behavior of a reactive system. In these systems, the *modes* (or *state*) of the software drive the behavior of the device. For example, in a car cruise controller, it could be a state machine describing how the controller engages and disengages depending on a number of parameters and actions.

Existing approaches compile hierarchical state machines into "pure" dataflow formalism (such as Lustre). While this approach is rather general, it has the disadvantage that the structure of state machine gets lost in the translation. This can have crucial consequences for verification methods based on inductive arguments, such as *k-induction*[17] or property directed reachability[16], because the logical encoding ends up creating a state space with states that do not correspond to any state of the original state

*This work was partially supported by the ANR-INSE-2012 CAFEIN project and NASA Contract No. NNX14AI09G.

doi:10.4204/EPTCS.219.2

machine, and so are unreachable by the resulting transition system. These states are problematic because they typically lead to spurious counter-examples for the inductive step of the verification process.

In this paper, we propose a technique to faithfully compile hierarchical state machines expressed as automata in the Lustre language into modular Horn clauses. Our compilation technique preserves the structural and modal behavior of the system, making the analysis of such models more tractable. Specifically, this paper makes the following contributions:

- a state-preserving encoding of hierarchical state machines as pure clocked-dataflow models. This encoding is inspired by the work described in [4]. Our technique differ in how we encode the state of each automaton, which gives a more flexible encoding.
- a compilation of hierarchical state machines into modular Horn clauses.
- finally, an implementation of the proposed compilation in LUSTREC [11] – an open source compiler for Lustre programs.

The rest of the paper is structured as follows: in the next sub-section, we give an overview of the synchronous dataflow language Lustre. In Section 2 we describe the semantics of the hierarchical state machines that we consider in this paper. In Section 3 we describe our structure preserving compilation scheme. In Section 4 we illustrate the extensions of the compiler to handle the compilation of automata in to Horn clauses. Finally, in Section 5 we illustrate our compilation approach on a simple yet representative example.

1.1 Background

Synchronous languages are a class of languages proposed for the design of so called "reactive systems" – systems that maintain a permanent interaction with physical environment. Such languages are based on the theory of synchronous time, in which the system and its environment are considered to both view time with some "abstract" universal clock. In order to simplify reasoning about such systems, outputs are usually considered to be calculated instantly [2]. Examples of such languages include Esterel [3], Signal [1] and Lustre [5, 12]. In this paper, we will concentrate on the latter. Lustre combines each data stream with an associated clock as a way to discretize time. The overall system is considered to have a universal clock that represents the smallest time span the system is able to distinguish, with additional, coarser-grained, user-defined clocks. Therefore the overall system may have different subsections that react to inputs at different frequencies. At each clock tick, the system is considered to evaluate all streams, so all values are considered stable for any actual time spent in the instant between ticks. A stream position can be used to indicate a specific value of a stream in a given instant, indexed by its clock tick. A stream at position 0 is in its initial configuration. Positions prior to this have no defined stream value. A Lustre program defines a set of equations of the form: $\mathsf{y}_1, \ldots, \mathsf{y}_n = \mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_m)$ where y_i are output or local variables and x_i are input variables. Variables in Lustre are used to represent individual streams and they are typed, with basic types including streams of *Real* numbers, *Integers*, and *Booleans*. Lustre programs and subprograms are expressed in terms of *Nodes*. Nodes directly model subsystems in a modular fashion, with an externally visible set of inputs and outputs. A *node* can be seen as a mapping of a finite set of input streams (in the form of a tuple) to a finite set of output streams (also expressed as a tuple). The *top node* is the main node of the program, the one that interface with the environment of the program and can never be called by another node.

At each instant t, the node takes in the values of its input streams and returns the values of its output streams. Operationally, a node has a cyclic behavior: at each cycle t, it takes as input the value of

```
type run_mode = enum { Start, Stop };

function switch (mode_in : run_mode) returns (mode_out : run_mode);
let mode_out = if mode_in = Start then Stop else Start; tel

node count (tick:bool) returns (seconds:int);
let seconds = 0 -> pre seconds + 1; tel

node stopwatch (tick:bool; start_stop:bool; reset:bool) returns (seconds : int);
var run : run_mode clock;
let run = Stop -> if start_stop then switch(pre run) else pre run;
    seconds = merge run (Start -> count(tick when Start(run)) every reset)
                        (Stop -> (0 -> pre seconds) when Stop(run));
tel
```

Figure 1: A simple Lustre program.

each input stream at position or instant t, and returns the value of each output stream at instant t. This computation is assumed to be immediate in the computation model. Lustre nodes have a limited form of memory in that, when computing the output values they can also look at input and output values from previous instants, up to a finite limit statically determined by the program itself.

Typically, the body of a Lustre node consists of a set of definitions, stream equations of the form $x = t$ (as seen in Figure 1) where x is a variable denoting an output or a locally defined stream and t is an expression, in a certain stream algebra, whose variables name input, output, or local streams. More generally, x can be a tuple of stream variables and t an expression evaluating to a tuple of the same type. Most of Lustre's operators are point-wise lifting to streams of the usual operators over stream values. For example, let $x = [x_0, x_1, \ldots]$ and $y = [y_0, y_1, \ldots]$ be two integer streams. Then, $x + y$ denotes the stream $[x_0 + y_0; x_1 + y_1, \ldots]$; an integer constant c, denotes the constant integer stream $[c, c, \ldots]$. Two important additional operators are a unary shift-right operator *pre* ("previous"), and a binary initialization operator $\rightarrow$ ("followed by"). The first is defined as $pre(x) = [u, x_0, x_1, \ldots]$ with the value u left unspecified. The second is defined as $x \rightarrow y = [x_0, y_1, y_2, \ldots]$. Syntactical restrictions on the equations in a Lustre program guarantee that all its streams are well defined: e.g. forbidding recursive definitions hence avoiding algebraic loops.

Figure 1 illustrate a simple *stopwatch* example using Lustre *enumerated clocks* and *node reset*. Enumerated clocks are an advanced form of the traditional Lustre clocks. They allow to sample a value of a flow depending on the value of a clock. For example, the expression "tick **when** Start(run) " denotes a signal that is only defined when the clock flow run has value Start. The sampled flows can be gathered together using the **merge** operator as in the definition of variable "seconds" in node stopwatch. Moreover, a node call can be reset to its initial state when a given boolean condition is set to true. For example, in Figure 1 the expression "count (..) **every** reset" will return the initial state of the node count. The function *switch* is a memoryless node, hence is declared with the keyword *function*.

2 Automaton as hierarchical state machines

Synchronous semantics of hierarchical state machines and their compilation to imperative code has been investigated in different articles, e.g. [14, 15, 22, 13], which resulted in a vast number of different incompatible semantics. Furthermore, the challenges of mixing state machines and dataflow formalism has also been the subject of intensive studies, from [19] to [7, 6]. In our setting, we follows the approach

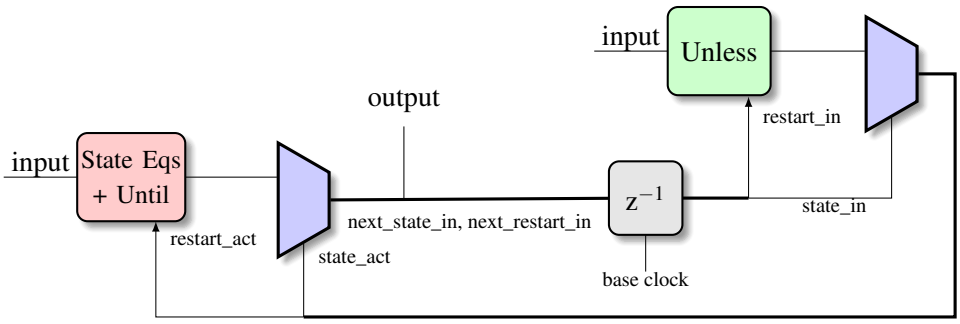


Figure 2: Automaton as a pure dataflow

developed in [7, 6], which is to the best of our knowledge the most disciplined and simple approach. This technique is also implemented in the commercial KCG Scade/Lustre compiler [9].

Informally, the modular compilation scheme developed in [7, 6] is enforced at the expense of raw (and somehow undesired) expressivity, disallowing for instance transitions that go through boundaries of hierarchical state machines or the firing of an unbounded number of transitions per instant (e.g. in Matlab Simulink and Stateflow). For instance, in Figure 2, at each instant, two pairs of variables are computed: a putative *state_in* and an actual state *state_act* and also, for both states, two booleans *restart_in* and *restart_act*, that tell whether their respective state equations should be reset before execution. The actual state is obtained via a strong (`unless`) transition from the putative state, whereas the next putative state is obtained via a weak (`until`) transition from the actual state. Only the actual state equations are executed at each instant. Finally, a reset function is driven by the `restart/resume` keyword switches. As transition-firing conditions may have their own memories, they can be reset if needed before being evaluated. Specifically, `unless` conditions are reset according to *restart_in*, whereas `until` and state equations altogether are reset according to *restart_act*. To recapitulate, a transition is evaluated as follows: unless conditions of the initial active state – the putative state – are evaluated. In case of a valid one, we jump to the associated state. Then, the state equations are evaluated: either the ones of the putative state in case of no unless transitions activated, or the ones of the new state obtained. Then, as a last step, the until transition of the active state are evaluated and characterize the next state for the following transition. At most one unless and one until transitions are evaluated, in this order, at each time step.

Our approach, builds on top of the aforementioned compilation scheme. In our setting, we promote the computation of strong transition, state equations and weak transition to independent auxiliary Lustre nodes. This allows a certain flexibility: (i) independent scheduling and optimization of different state equations; (ii) addition of code contracts to different states. Those features are not supported by the commercial KCG suite. Another benefit of our approach is that we don't modify state equations to take clock constraints, nor local variables or the reset operation. Local state information is only recovered through clock calculus and is not structural any more, as generated code may be optimized and scattered. We rather only encapsulate state equations in new node definitions and generate new equations for calling these nodes, greatly facilitating the management of local state invariants for instance. Yet, this comes at the expense of a rather limited loss in expressivity, due to possible causality issues[1]. Note that inlining these auxiliary nodes is already an available option that fully recovers the original semantics.

We illustrate in Listings. 1a, 1b and 1c, the differences between our approach and [7], from a user's

[1]We recall that the classical causality analysis in modern Lustre doesn't cross boundaries of nodes, hence the conservative rejection of some correct programs.

```
node failure (i:int) returns (o1, o2:int);
let
  (o1, o2) = if i = 0
             then (o2, i)
             else (i, o1);
tel
```

(a) Scheduling failure

```
node solution (i:int) returns (o1, o2:int);
let
  automaton condition
  unless i <> 0 resume KO
  state OK:
  let
    (o1, o2) = (o2, i);
  tel
  state KO:
  unless i = 0 resume OK
  let
    (o1, o2) = (i, o1);
  tel
tel
```

(b) Automaton based solution

```
node triangle (r:bool) returns (o:int);
let
  automaton trivial
  state One:
  unless r || pre o = 100
  let
    o = 0 -> 1 + pre o;
  tel
tel
```

(c) Causality issues

Listing 1: Examples comparing our approach with the one developed in [7]

viewpoint. Example 1a is a typical program that cannot be statically scheduled and produces a compilation error in both approaches. A solution may be devised as in Example 1b, using an automaton to encode the boolean switch $i = 0$. Even if scheduling is done prior to other static analyses (and thus is unaware of exclusive automaton states for instance), we succeed in generating a correct code whereas KCG fails. Example 1c is non-causal and won't compile if we remove the pre occurring in the `unless` clause. But if we keep it, KCG will handle it correctly whereas our causality analysis will reject this program. Generally speaking, we forbid `unless` clauses that would refer to putative state memories (such as o). Accepting these clauses appear problematic or at least confusing as it makes the putative state visible and distinct from the actual state, thus duplicating state variables.

As a result our encoding is not strictly comparable with Scade or Lustre v6 automaton. For instance, we are unable to type check and compile automaton with memories within unless conditions. This is possible in our setting. In summary, our encoding does not flatten the automaton into a single Lustre node but preserves the structure by associating a Lustre node for each automaton state. This structure preserving encoding enables us to analyze these models and compute local invariants associated to automaton states.

3 Synchronous dataflow programs as Horn clauses

In [10] we have developed a compilation technique that translate Lustre programs into modular Horn clauses. In order to accommodate the compilation of automaton we have updated such compilation scheme. In this section, we briefly describe the different stages of the compilation process. For a formal treatment of the compilation stages the reader can refer to [10]. Figure 3 illustrate the compilation stages implemented in LUSTREC.

Automaton compilation. The first phase, which is new w.r.t. [10], compiles the hierarchical states machines as pure dataflow expression in Lustre. A detailed description of this phase will be presented in Section. 4.

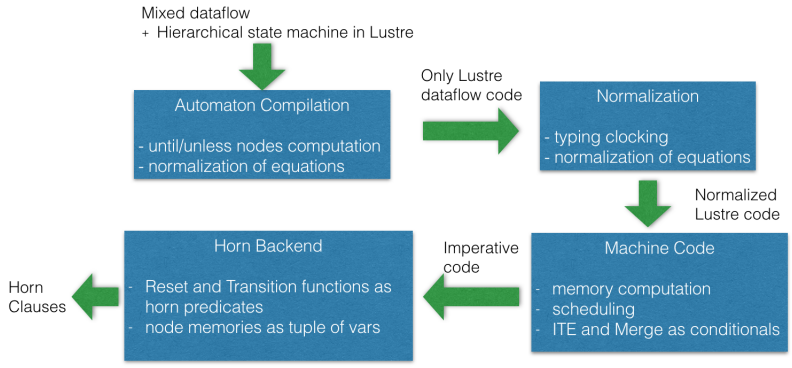


Figure 3: Compilation Stages

Normalization. This phase infers types and clock for each signal. Each expression is recursively normalized: node calls, **pre** constructs, tuples, ...are defined as fresh variables. Each function call $foo(args)$ is associated to a unique identifier $foo^{uid}(args)$. Once normalized, no function calls occurs within expressions, nor arrows nor definitions of memories through a `pre` construct.

Machine code. In the main compiler process, intended to generate embedded code, the next phase generates the machine (or imperative) code: its flows definitions are replaced by an ordered sequence of imperative statements. The state of a node instance is characterized by its memories, i.e. expressions defining memories such as x = **pre** e, and the node instances that appear in its expressions such as x = foo(...).

Generation of machine code. At this stage a machine code is generated. This amount to replace the flow definitions with an ordered sequence of imperative statements. The state of a node instance is characterized by its memories, i.e. expressions defining memories such as $x = pre\ e$, and the node instances that appear in its expressions such as $x = foo(\ldots)$. This tree-like characterization of a node state enable a modular definition of a state as a tree of local memories.

Definition 1 (Node memories and instances)**.** *Let f be a Lustre node with normalized equations eqs. Then we define its set of memories and node callee instances as:*

$$\begin{aligned} Mems(f) = &\ \{x \mid x = pre\ \_ \in eqs\} \\ Insts(f) = &\ \{(foo, uid) \mid \_ = foo^{uid}(\_) \in eqs\} \end{aligned}$$

The *follow by* ($\rightarrow$) operator is interpreted as a node instance of a generic polymorphic node arrow as illustrated in Listing 2a. Therefore the initial state for a node associated to all its arrow instances and all its child arrow instances the value true to the memory init . Similarly the activation of a node reset using the operator *every* modifies the state of this instance of the node foo by resetting its arrow init variable to their initial value true, e.g. e = foo (...) **every** ClockValue(clock_var). Figure 4 shows the computed memories of the node in Listing 2b.

Horn backend. At this stage the Horn clauses are generated. The hierarchy of memories and node instances are flattened and modeled as a tuple of memories. We denote as $\texttt{state}^{\texttt{label}}$ (f, uid) the tuple of variables denoting the state of the instance uid of a node f. Different labels are used to differentiate

```
node arrow (e1, e2: 'a) returns (out: 'a)
var init: bool;
let
  init = true -> false;
  out = if init then e1 else e2;
tel
```

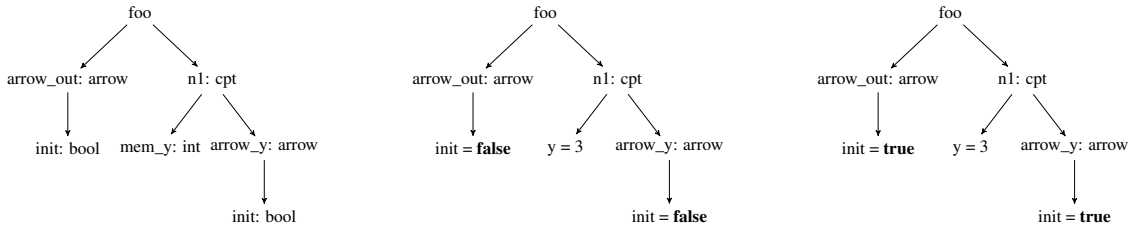
(a) Polymorphic arrow node

```
node cpt (z: bool) returns (y: int);
let y = 0 -> if z then 0 else pre y + 1; tel

node foo (z: bool) returns (out: int)
let out = 1 -> foo(z); tel
```

(b) Simple example with two nodes

Listing 2: Memories in Lustre



(a) Tree type: memories and instances (b) Example of state for node foo (c) State (4b) after reset

Figure 4: Memory trees and reset

between different versions of the same variable. We use the labels *c* and *n* to denote the (c)urrent and (n)ext value of a memory x: x^c and x^n. The internal node *arrow* is fitted with a specific *reset* rule:

```
rule (=> (= init^n true) (arrow_reset (init^c, init^n)))
```

Note that its state is only defined by the init variable. In the proposed encoding, the predicates defining the program semantics should enable the reset of a node state as performed in Fig. 4c. This leads to the following encoding for the reset function:

```
rule (=> ( ⋀_{mem ∈ Mems(f)} (= mem^n mem^c)  ⋀_{(g, uid) ∈ Inst(f)} g_reset (state^c (g,guid), state^n (g,guid)))
         (f_reset (state^c(f,uid), state^n(f,uid))))
```

The collecting semantics definition of [10] is modified to rely on f_reset instead of f_init. It builds the set of reachable states (*Reach*):

```
rule (=> (f_reset (state^c(f,uid), state^n(f,uid))) (Reach (state^n(f,uid))))
rule (=> (and (f_step (inputs, outputs, state^c(f,uid), state^n(f,uid)))
              (Reach (state^c(f,uid))))
         (Reach (state^n(f,uid))))
```

4 Compilation of automaton

In this section, we describe the compilation scheme from automaton to modular Horn clauses. This is performed in two stages: (i) compilation of automaton into clocked expressions and (ii) compilation of clocked expressions into Horn clauses.

4.1 From automaton to clocked expressions

We denote with $ReadEqs_i$ and $WriteEqs_i$ the set of read and write variables occurring in equations of an automaton state S_i. We also denote as $ReadUnless_i$ and $ReadUntil_i$ the set of variables in `unless` and `until` clauses.

```
node nd (inputs) returns (outputs);
var locals
let
  other_equations
  automaton aut
  ...
  state S_i:
  ...
  unless (sc_j,sr_j,SS_j)
  ...
  var locals_i
  let
    equations_i
  tel
  ...
  until (wc_j,wr_j,WS_j)
  ...
tel
```

(a) Automaton skeleton

```
type aut_type = enum { S_1, ..., S_n };
...
node S_i_unless (ReadUnless_i)
returns (restart_act : bool,
         state_act : aut_type clock);
let
  (restart_act, state_act) =
    if sc_1 then (sr_1, SS_1) else
    if sc_2 then (sr_2, SS_2) else
    ...
    (false, S_i);
tel

node S_i_handler_until (ReadEqs_i ∪ ReadUntil_i)
returns (restart_in : bool,
         state_in : aut_type clock,
         WriteEqs);
var locals_i
let
  (restart_in, state_in) =
    if wc_1 then (wr_1, WS_1) else
    if wc_2 then (wr_2, WS_2) else
    ...
    (false, S_i);
  equations_i
tel
```

(b) Clocked expression as new nodes.

Listing 3: Automaton in Lustre and their representation as clocked expressions in Lustre nodes.

Our compilation scheme from automaton to clocked expressions follows Figure 2 and is applied to a generic automaton such as the one described in Figure 3a (node *nd*). As illustrated in Listing 3b, the variables *state_act* and *state_in* are modelled as clocks of enumerated type. Also, two new nodes are introduced for each of the automaton state: one to express the semantics of state equations; and another one to capture the *weak* and *strong* transitions (as explained in Section 2).

Figure 4 illustrate the compiled node *c_nd* that replace the original automaton description of node *nd*. Evaluation of each single node call embedding state equations and transitions only takes place when its corresponding clock is active; this is done via "**when** *Value*(*clock*)" sampling operators applied to all node arguments.

All the node calls that corresponds to the global evaluation of the automaton are then gathered in two **merge** constructs, which are driven by the putative state clock *state_in* (for strong transitions) and the actual state clock *state_act* (for weak transitions and state equations).

4.2 Compiling clocked expressions into modular Horn clauses

Once the automaton structure as been compiled into clocked expression, the second step is to encode them as Horn clauses. Here, we use the Horn clause format introduced in Z3 [16], where (`rule` *expr*) universally quantify the free variables of the SMT-LIB expression *expr*. At this level, the challenges are to be able to express within the Horn formalism the following concepts: (i) the clock's feature of Lustre, (ii) the reset functionality of a node, (iii) the declaration of enumerated clocks, (iv) clocked expression with the **when** operator, (v) merge of clocked expressions and reset of node state on conditionals with the **every** operator. In the following we illustrate how we capture using Horn clauses the above mentioned concepts.

```
node c_nd (inputs) returns (outputs);
var locals;
    aut_restart_in, aut_next_restart_in, aut_restart_act : bool;
    aut_state_in, aut_next_state_in, aut_state_act : aut_type clock;
let
  ...
  (aut_restart_in, aut_state_in) =
    (false, S1) -> pre (aut_next_restart_in, aut_next_state_in);
  (aut_restart_act, aut_state_act) = merge aut_state_in
     ...
     (Si -> Si_unless((ReadUnlessi) when Si(aut_state_in)) every aut_restart_in)
     ...
  (aut_next_restart_in, aut_state_next_in, WriteEqs) = merge aut_state_act
     ...
     (Si -> Si_handler_until((ReadEqsi ∪ ReadUntili) when Si(aut_state_act)) every aut_restart_act)
    ...
tel
```

Listing 4: Compiled node *c_nd* from node *nd* in Figure 3a.

Clock values are defined as regular enumerated type in SMT-LIB format:

```
(declare-datatypes () ((clock_type Start Stop)))
```

Combination of **merge** and **when** operator are required for the clock calculus (i.e. clock typing) but are ignored when generating the final code. Merge constructs act as a switch-case statement over well-clocked expression. For example, the following well-clocked Lustre expression:

```
seconds = merge run (Start -> x when Start(run)) (Stop -> y when Stop(run))
```

is translated to the imperative switch-case expression:

```
switch (run) { case Start : e = x; break;
               case Stop  : e = y; break }
```

Since each case definition is purely functional, this can be directly expressed as the following constraint:

```
(and (=> (= run Start) (= e x))
     (=> (= run Stop)  (= e y)))
```

The next item is to capture the reset of a node's state using the **every** operator,
e.g. count (x) **every** condition. During the compilation process, such expression generate a machine code instruction:

```
if (condition) { Reset(count, uid) };
```

which gets translated to an imperative statement:

```
if (condition) { count_reset(state_count_uid) } else {};
```

where *state_count_uid* is a *struct* that denotes a node's state instance. s This conditional statement will perform a side-effect update of the memory state and impact the computation of the next state and outputs. How do we capture this in the Horn encoding? Let us first look how we encode a step transition in Horn clauses. A step transition is basically a relationship (i.e. a predicate) between inputs, outputs, previous state and next state.

```
relationship(inputs, outputs, old_state, new_state) = (and (...))
```

Typically this relationship would be used to define the step transition as follows:

```
rule (=> (relationship(inputs, outputs, state^c(f,uid), state^n(f,uid)))
         (f_step (inputs, outputs, state^c(f,uid), state^n(f,uid))))
```

```
node auto (x:bool) returns (out:bool);
let
  automaton four_states
  state One :
  let
   out = false;
  tel until true restart Two
  state Two :
  let
   out = false;
  tel until true restart Three
  state Three :
  let
   out = true;
  tel until true restart Four
  state Four :
  let
   out = false;
  tel until true restart One
tel
```

(a) Automaton-based counter

```
node greycounter (x:bool) returns (out:bool);
var a,b:bool;
let
  a = false -> not pre(b);
  b = false -> pre(a);
  out = a and b;
tel
```

(b) Boolean-based counter

```
node intloopcounter (x:bool) returns (out:bool);
var time: int;
let
  time = 0 -> if pre(time) = 3 then 0
              else pre time + 1;
  out = (time = 2);
tel
```

(c) Integer-based counter

Listing 5: Automaton-(5a), Boolean-(5b) and Integer-(5c) based implementation of a 2-bit counter.

For resetting the node's state, the value used for the state is $\mathsf{f\_reset}(\mathsf{state}^c(\mathsf{f},\mathsf{uid}))$ (instead of $\mathsf{state}^c(\mathsf{f},\mathsf{uid})$). In addition to the two state labels used to denote current c and next value x, we introduce an intermediate label i. In case of a transition without reset, the intermediate version would be directly defined as the current one.

```
rule (=> (and (= (state^i(f,uid)) (state^c(f,uid)))
              (relationship(inputs, outputs, state^i(f,uid), state^n(f,uid))))
         (f_step (inputs, outputs, state^c(f,uid), state^n(f,uid))))
```

Finally, the reset function is encoded as follows:

```
rule (=> (and (= (state^i(f,uid))
                 (if condition then f_reset(state^c(f,uid)) else state^c(f,uid)))
              (relationship(inputs, outputs, state^i(f,uid), state^n(f,uid))))
         (f_step (inputs, outputs, state^c(f,uid), state^n(f,uid))))
```

5 Example of encoding

As an example of the proposed compilation process, we consider a simple Lustre program that compares three implementations of a 2-bit counter: a low-level Boolean implementation, a higher-level implementation using integers and an automaton based counter. The `greycounter` node (cf. Fig. 5b) internally repeats the sequence $ab = \{00,01,11,10,00,...\}$ indefinitely, while the `integercounter` node (cf. Fig. 5c) repeats the sequence $time = \{0,1,2,3,0,\dots\}$. The automaton based node (cf. Fig. 5a) is a state machines with 4 states and it basically alternates between them.
In the first phase of our compilation scheme a clock is generated to encode the automaton states:

```
type auto_ck = enum {One, Two, Three, Four };
```

Each automaton state is associated to a stateless function describing respectively its strong (*unless*) transitions and its weak (*until*) ones.

```
node auto (x: bool) returns (out: bool)
  var mem_restart: bool; mem_state: auto_ck;
    four_restart_in: bool; four_state_in: auto_ck; four_out: bool ;
    four_restart_act: bool; four_restart_in: auto_ck;
    ... -- similar declarations for other states
    next_restart_in: bool; restart_in: bool;
    restart_act: bool; next_state_in: auto_ck;
    state_in: auto_ck clock; state_act: auto_ck clock;
 let
   restart_in, state_in = ((false,One) -> (mem_restart,mem_state));
   mem_restart, mem_state = pre (next_restart_in,next_state_in);
   next_restart_in, next_state_in, out =
            merge state_act (One -> ...) (Two -> ...)  (Three -> ...)
                            (Four -> (four_restart_in, four_state_in, four_out));
   four_restart_in, four_state_in, four_out =
   Four_handler_until (     restart_act when Four(state_act),
                            state_act when Four(state_act))
                      every (restart_act);
    ... -- similar definitions for other states
   restart_act, state_act = merge state_in (One -> ...)
                                           (Two -> ...)
                                           (Three -> ...)
                                  (Four -> (four_restart_act, four_state_act));
   four_restart_act, four_state_act =
     Four_unless (     restart_in when Four(state_in),
                       state_in when Four(state_in))
                 every (restart_in);
   ... -- similar definitions for other states
  tel
```

Listing 6: Generated Lustre code without automaton.

To keep the presentation simpler we present the encoding of the state *Four*:

```
function Four_handler_until (restart_act: bool; state_act: auto_ck)
   returns (restart_in: bool; state_in: auto_ck; out: bool)
 let -- encodes the next state, here One
     restart_in, state_in = (true,One);
     out = false;-- returns true in the handler for state Three
 tel
```

The handler and the *until* function assigns the next state to the state *One* and require the node to be restarted. Listing 6 shows the generated Lustre node without an automaton.
The next stage of the compiler produces the Horn clauses. Enumerated type enable the declaration of clock's values:

```
(declare-data auto_ck () ((auto_ck One Two Three Four)))
```

The functions for *until* and *unless* are defined as Horn predicates (*Four_handler_until* and *Four_unless* respectively) in the following way:

```
(declare-rel Four_handler_until (Bool auto_ck Bool auto_ck Bool))
(rule (=> (and (= out false) (= state_in One) (= restart_in true))
          (Four_handler_until restart_act state_act restart_in  state_in out)))
(declare-rel Four_unless (Bool auto_ck Bool auto_ck))
(rule (=> (and (= state_act state_in) (= restart_act restart_in))
          (Four_unless restart_in state_in restart_act state_act)))
```

Finally the reset (*auto_reset*) and step (*auto_step*) predicates are defined as follows respectively:

```
(rule (=> (and (= mem_restart_m mem_restart_c) (= mem_state_m mem_state_c)
               (= arrow.init_m true))
      (auto_reset mem_restart_c mem_state_c arrow.init_c
                  mem_restart_m mem_state_m arrow.init_m)))

(rule (=>
    (and (= arrow.init_m arrow.init_c)
         (= arrow.init_x false) -- update of arrow state
    (and (=> (= arrow.init_m true) -- current arrow is first
             (and (= state_in One)
                  (= restart_in false)))
         (=> (= arrow.init_m false) -- current arrow is not first
             (and (= state_in mem_state_c)
               (= restart_in mem_restart_c))))
     (and (=> (= state_in Four) -- unless block for automaton state Four
          (and (Four_unless restart_in state_in four_restart_act four_state_act)
               (= state_act four_state_act)
               (= restart_act four_restart_act)))
              ...) -- similar definition for other states
     (and (=> (= state_act Four) -- handler and until block for state Four
          (and (Four_handler_until restart_act state_act
                                   four_restart_in four_state_in four_out)
              (= out four_out)
              (= next_state_in four_state_in)
              (= next_restart_in four_restart_in)))
               ...) -- similar definition for other states
          (= mem_state_x next_state_in) -- next value for memory mem_state
          (= mem_restart_x next_restart_in)) -- next value for memory mem_restart
(auto_step x    -- inputs
           out  -- outputs
           mem_restart_c mem_state_c arrow.init_c    -- old state
           mem_restart_x mem_state_x arrow.init_x))) -- new state
```

Once the Horn clauses are generated, a Horn clause solver can be used to perform verification and/or testing. For example, we used Spacer [18] to prove that the three implementation of the 2-bit counters behaves the same (i.e. each implementation outputs the stream true).

6 Conclusion

In this paper, we proposed a new compilation scheme to faithfully compile a mix of dataflow formalism and hierarchical state machines into modular Horn clauses. Our approach compile hierarchical state machines expressed as automata in the Lustre language into modular Horn clauses. The compilation technique preserves the structural and modal behavior of the system which makes the analysis of such models more tractable. The proposed approach is implemented in LUSTREC– an open source Lustre compiler. Once the modular Horn clauses are generated, automated reasoning tools like Spacer [18] can be used to reason about properties. In the future, we plan to evaluate our compilation scheme on larger industrial case studies.

References

[1] P. Amagbégnon, L. Besnard & P. Le Guernic (1995): *Implementation of the Data-flow Synchronous Language SIGNAL*. In: *Conference on Programming Language Design and Implementation*, ACM Press, pp. 163–173, doi:10.1145/207110.207134.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic & Robert de Simone (2003): *The synchronous languages 12 years later*. *Proceedings of the IEEE* 91(1), pp. 64–83, doi:10.1109/JPROC.2002.805826.

[3] G. Berry & G. Gonthier (1992): *The ESTEREL synchronous programming language: design, semantics, implementation*. *Sci. Comput. Program.* 19(2), pp. 87–152, doi:10.1016/0167-6423(92)90005-V.

[4] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon & Marc Pouzet (2008): *Clock-directed modular code generation for synchronous data-flow languages*. In Krisztián Flautner & John Regehr, editors: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, ACM, pp. 121–130, doi:10.1145/1375657.1375674.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs & John Plaice (1987): *Lustre: A Declarative Language for Programming Synchronous Systems*. In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, ACM Press, pp. 178–188, doi:10.1145/41625.41641.

[6] Jean-Louis Colaço, Grégoire Hamon & Marc Pouzet (2006): *Mixing signals and modes in synchronous data-flow systems*. In Sang Lyul Min & Wang Yi, editors: *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, ACM, pp. 73–82, doi:10.1145/1176887.1176899.

[7] Jean-Louis Colaço, Bruno Pagano & Marc Pouzet (2005): *A conservative extension of synchronous data-flow with state machines*. In Wayne Wolf, editor: *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, ACM, pp. 173–182, doi:10.1145/1086228.1086261.

[8] DO-178b: *Software Considerations in Airborne Systems and Equipment Certification*.

[9] Inc. Esterel Technologies: *SCADE*. Available at `http://www.esterel-technologies.com/products/scade-suite/`.

[10] Pierre-Loïc Garoche, Arie Gurfinkel & Temesghen Kahsai (2014): *Synthesizing Modular Invariants for Synchronous Code*. In Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko & Valerio Senni, editors: *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014.*, *EPTCS* 169, pp. 19–30, doi:10.4204/EPTCS.169.4.

[11] Pierre-Loïc Garoche, Temesghen Kahsai & Xavier Thirioux: *LustreC*. Available at `https://github.com/coco-team/lustrec`.

[12] N. Halbwachs, P. Caspi, P. Raymond & D. Pilaud (1991): *The synchronous dataflow programming language LUSTRE*. In: *Proceedings of the IEEE*, pp. 1305–1320, doi:10.1109/5.97300.

[13] Grégoire Hamon & John M. Rushby (2004): *An Operational Semantics for Stateflow*. In Michel Wermelinger & Tiziana Margaria, editors: *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004 Barcelona, Spain, March 29 - april 2, 2004, Proceedings*, *Lecture Notes in Computer Science* 2984, Springer, pp. 229–243, doi:10.1007/978-3-540-24721-0_17.

[14] David Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *Sci. Comput. Program.* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[15] David Harel & Michal Politi (1998): *Modeling Reactive Systems with Statecharts: The Statemate Approach*, 1st edition. McGraw-Hill, Inc., New York, NY, USA.

[16] Kryštof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In Alessandro Cimatti & Roberto Sebastiani, editors: *Theory and Applications of Satisfiability Testing – SAT 2012*, *LNCS* 7317, pp. 157–171, doi:10.1007/978-3-642-31612-8_13.

[17] Temesghen Kahsai & Cesare Tinelli (2011): *PKind: A parallel k-induction based model checker*. In Jiri Barnat & Keijo Heljanko, editors: *Proceedings 10th International Workshop on Parallel and Distributed*

Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011., *EPTCS* 72, pp. 55–62, doi:10.4204/EPTCS.72.6.

[18] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki & Edmund M. Clarke (2013): *Automatic Abstraction in SMT-Based Unbounded Software Model Checking*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, *Lecture Notes in Computer Science* 8044, Springer, pp. 846–862, doi:10.1007/978-3-642-39799-8_59.

[19] Florence Maraninchi & Yann Rémond (1998): *Mode-automata: About modes and states for reactive systems*. In Chris Hankin, editor: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 1381, Springer Berlin Heidelberg, pp. 185–199, doi:10.1007/BFb0053571.

[20] Inc. The MathWorks: *Simulink*. Available at `http://www.mathworks.com/products/simulink/`.

[21] Inc. The MathWorks: *Stateflow*. Available at `http://www.mathworks.com/products/stateflow/`.

[22] Andrew C. Uselton & Scott A. Smolka (1994): *A Compositional Semantics for Statecharts using Labeled Transition Systems*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, *Lecture Notes in Computer Science* 836, Springer, pp. 2–17, doi:10.1007/BFb0014994.